46장 제너레이터와 async/await

46.1 제너레이터란?

코드 블록의 실행을 일시 중지했다가 필요한 시점에 재개할 수 있는 특수한 함수

1. 제너레이터 함수는 함수 호출자에게 함수 실행의 제어권을 양도할 수 있다.
2. 제너레이터 함수는 함수 호출자와 함수의 상태를 주고받을 수 있다.
3. 제너레이터 함수를 호출하면 제너레이터 객체를 반환한다.

46.2 제너레이터 함수의 정의

- `function*` **키워드로 선언한다.**
- 하나 이상의 `yield 표현식`을 포함한다.

```
// 제너레이터 함수 선언문
function* genDecFunc() {
  yield 1;
}

// 제너레이터 함수 표현식
const genExpFunc = function* () {
  yield 1;
}

// 제너레이터 메서드
const obj = {
  * genObjMethod() {
    yield 1;
  }
}

// 제너레이터 클래스 메서드
class MyClass {
  * genClsMethod() {
```

```
    yield 1;
  }
}
```

- 제너레이터 화살표 함수로 정의 불가

```
const genArrowFunc = * () => {
  yield 1;
}// syntaxError: Unexpected token '*'
```

- 제너레이터 new 연산자와 함께 생성자 함수로 호출 불가

```
function* genFunc() {
  yield 1;
}

new genFunc(); // TypeError: genFunc is not a constructor.
```

46.3 제너레이터 객체

- **제너레이터 함수를 호출하면 일반 함수처럼 함수 코드 블록을 실행하는 것이 아니라 제너레이터 객체를 생성해 반환**
- **제너레이터 함수가 반환한 제너레이터 객체는 이터러블이면서 동시에 이터레이터**
- 제너레이터 객체는 `Symbol.iterator` 메서드를 상속받는 이터러블이면서,
  value, done 프로퍼티를 갖는 이터레이터 리절트 객체를 반환하는
  next 메서드를 소유하는 이터레이터

```
// 제너레이터 함수
function* genFunc() {
  yield 1;
  yield 2;
  yield 3;
}

// 제너레이터 함수를 호출하면 제너레이터 객체를 반환한다.
```

```
const generator = genFunc();

// 제너레이터 객체는 이터러블이면서 동시에 이터레이터이다.
// 이터러블은 Symbol.iterator 메서드를 직접 구현하거나 프로토타입 체
인을 통해 상속받은 객체다.
console.log(Symbol.iterator in generator); // true
// 이터레이터는 next 메서드를 갖는다.
console.log('next' in generator) // true
```

- 제너레이터 객체는 next 메서드를 갖는 이터레이터이지만

  이터레이터에는 없는 return, throw 메서드를 갖는다.
- 제너레이터 객체의 세 개의 메서드 호출 시 동작 과정
  - **next 메서드를 호출**하면 제너레이터 함수의 yield 표현식까지 코드 블록을 실행하고 yield된 값을 value 프로퍼티 값으로, false를 done 프로퍼티 값으로 갖는

    이터레이터 리절트 객체를 반환한다.
  - **return 메서드를 호출**하면 인수로 전달받은 값을 value 프로퍼티 값으로,

    true를 done 프로퍼티 값으로 갖는 이터레이터 리절트 객체를 반환한다.
  - **throw 메서드를 호출**하면 인수로 전달받은 에러를 발생시키고 undefined를 value 프로퍼티 값으로, true를 done 프로퍼티 값으로 갖는 이터레이터 리절트 객체를 반환한다.

```
function* genFunc() {
  try {
    yield 1;
    yield 2;
    yield 3;
  } catch (e) {
    console.error(e);
  }
}

const generator = genFunc();

console.log(generator.next()); // {value: 1, done: false}
```

```
console.log(generator.return('End!')); // {value: 'End!', d
one: return}
```

46.4 제너레이터 일시 중지와 재개

yield 키워드는 제너레이터 함수의 실행을 일시 중지 시키거나 yield 키워드 뒤에 오는 표현식의 평가 결과를 제너레이터 함수 호출자에게 반환한다.

```
function* genFunc() {
  const x = yield 1;

  const y = yield (x + 10);

  // 따라서 제너레이터에서는 값을 반환할 필요가 없으며
  // return은 종료의 의미로만 사용해야 한다.
  return x + y;
}

const generator = genFunc(0);

let res = generator.next();
console.log(res); // {value: 1, done: false}

res = generator.next(10);
console.log(res); // {value: 20, done: true}

res = generator.next(20);
console.log(res); // {value: 30, done: true}
```

46.5 제너레이터의 활용

46.5.1 이터러블의 구현

- 무한 피보나치 수열 생성 함수

```
// 무한 이터러블을 생성하는 함수
const infiniteFibonacci = (function () {
  let [pre, cur] = [0, 1];

  return {
    [Symbol.iterator]() { return this; },
    next() {
      [pre, cur] = [cur, pre + cur];
      // 무한 이터러블이므로 done 프로퍼티 생략한다.
      return { value: cur };
    }
  }
}());

// infiniteFibonacci는 무한 이터러블이다.
for (const num of infiniteFibonacci) {
  if (num > 10000) break;
  console.log(num); // 1 2 3 5 8...
}
```

- 제너레이터를 사용한 무한 피보나치 수열 생성 함수

```
// 무한 이터러블을 생성하는 제너레이터 함수
const infiniteFibonacci = (function* () {
  let [pre, cur] = [0, 1];

  while(true) {
    [pre, cur] = [cur, pre + cur];
    yield cur;
  }
}());
```

46.5.2 비동기 처리

제너레이터 함수는 프로미스를 사용한 비동기 처리를 동기 처리 처럼 구현할 수 있다.

```
const fetch = require('node-fetch');
```

```
// 제너레이터 실행기
const async = generatorFunc => {
  const generator = generatorFunc(); // (2)

  const onResolved = arg => {
    const result = generator.next(arg); // (5)

    return result.done
     ? result.value // (9)
     : result.value.then(res -> onResolved(res)); // (7)
  };
  return onResolved; // (3)
};

(async(function* fetchTodo() { // (1)
    const url = '대충 url';

    const response = yield fetch(url); // (6)
    const todo = yield response.json(); // (8)
    console.log(todo);
})()); // (4)
```

46.6 async/await

- ECMAScript 2017(ES8)에서는 제너레이터보다 간단하고 가독성 좋게 비동기 처리를 동기 처리처럼 동작할 수 있는 `async/await`가 도입
- 프로미스 기반으로 동작하며 프로미스의 후속 처리 메서드 없이 마치 동기 처리처럼 프로미스가 처리 결과를 반환하도록 구현할 수 있다.

46.6.1 async 함수

```
// async 함수 선언문
async function foo(n) { return n; }
foo(1).then(v => console.log(v)); // 1

// async 메서드
```

```
const obj = {
  async foo(n) { return n; }
}
obj.foo(4).then(v => console.log(v)); // 4
```

- 클래스의 contructor 메서드는 async 메서드가 될 수 없다.
- 클래스의 constructor 메서드는 인스턴스를 반환해야 하지만

  async 함수는 언제나 프로미스를 반환해야 한다.

46.6.2 await 키워드

await 키워드는 프로미스가 **settled 상태(비동기 처리가 수행된 상태)**가 될 때까지 대기하다가 settled 상태가 되면 reslove한 처리 결과를 반환한다.

> 💡 모든 프로미스에 await 키워드를 사용하는 것은 주의해야한다.

46.6.3 에러 처리

콜 스택의 아래방향(실행 중인 컨텍스트가 푸시되기 직전에 푸시된 실행 컨텍스트 방향)으로 전파되기 때문에 에러 처리가 곤란하다.

하지만 비동기 함수의 콜백함수를 호출한 것은 비동기 함수가 아니기 때문에 try ... catch 문을 사용해 에러를 캐치할 수 없다.

하지만 `async/await` 에서 에러 처리는 try ... catch 문을 사용할 수 있다.

> 💡 async 함수 내에서 catch 문을 사용해서 에러 처리를 하지 않으면
> async 함수는 발생한 에러를 reject하는 프로미스를 반환한다.